

AFRL-IF-RS-TR-2005-383
Final Technical Report
November 2005



GINSU: GUARANTEED INTERNET STACK UTILIZATION

Trusted Information Systems, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. ARPS

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-383 has been reviewed and is approved for publication.

APPROVED: /s/

DAVID E. KRZYSIAK
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE NOVEMBER 2005	3. REPORT TYPE AND DATES COVERED Final Jun 01 – Apr 03	
4. TITLE AND SUBTITLE GINSU: GUARANTEED INTERNET STACK UTILIZATION			5. FUNDING NUMBERS C - F33615-01-C-1973 PE - 609199F PR - ARPS TA - NZ WU - OL	
6. AUTHOR(S) Roger Knobbe and Andrew Purtell				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Trusted Information Systems 3060 Washington Road Glenwood Maryland 21738			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2005-383	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: David E. Krzysiak/IFGA/(315) 330-7454/ David.Krzysiak@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) To design and implement an alternative IP host stack capable of guaranteeing availability of the stack to individual processes running on a computer. An IP host stack implementation is an important element of any networked host's operating system. Guaranteeing availability of network access requires a survivable host stack implementation as an alternative to today's IP host stack implementation.				
14. SUBJECT TERMS Communications Networks, Computer Architecture Data Links, Internet, Protocol Stacks				15. NUMBER OF PAGES 34
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

1. GINSU OVERVIEW	1
2. GINSU ARCHITECTURE	1
2.1 GINSU_COMMON	4
2.1.1 <i>Hooks</i>	4
2.1.2 <i>Maps</i>	8
2.1.3 <i>Resource Management</i>	9
2.1.4 <i>Logging</i>	12
2.2 GINSU_LOW	12
2.2.1 <i>Installing Resource Constraints</i>	14
2.2.2 <i>Startup Sequence</i>	19
2.3 THE DYNAMIC PACKET FILTER (DPF) API	19
2.4 QUEUE MANAGEMENT	20
2.5 SLICE SCHEDULING	21
2.5.1 <i>Accounting Practices</i>	22
2.5.2 <i>Resource Constraint Enforcement</i>	24
2.6 GINSU_PROC	26
3. DEMONSTRATION	28
4. FUTURE WORK	28
5. REFERENCES	29

LIST OF FIGURES

Figure 1: The High-level GINSU System Architecture.....	2
Figure 2: Module Stacking in the GINSU System.....	4
Figure 3: The GINSU Generic Resource Framework.....	10
Figure 4: The GINSU Task Structure	22
Figure 5: The GINSU Slice Structure	23
Figure 6: The GINSU Sock Structure	24

1. GINSU Overview

GINSU is a DARPA Fault Tolerant Networking, Focused Research Topic project. We improve the integrity and robustness of the Linux host operating system by isolating and monitoring traffic streams within the kernel. We allow an administrator to pre-allocate system resources across multiple axes, including process name, connection state, UNIX user or group identity, and source and/or destination endpoint addresses. GINSU monitors scarce resources, such as socket buffers, TCP control blocks, TCP Ports, CPU time, etc., across these axes and makes scheduling decisions with respect to the allotments of these resources and their actual use. GINSU does “early demultiplexing” of network traffic in order to determine the ultimate owner of network traffic. GINSU ensures that schedulable entities (network streams, processes, and protocols) are isolated from each other across all resource boundaries and guarantees that malicious or unanticipated levels of network traffic cannot compromise operating system and service integrity.

2. GINSU Architecture

GINSU presently supports the Linux open-source operating system, specifically, any distribution based on late versions of the 2.4-series kernel. On Linux, GINSU is implemented as a collection of loadable kernel modules. When GINSU modules are not loaded we do not impact normal Linux kernel functions at all. We set up GINSU-specific state and bring all managed network resources under the GINSU regime once our modules are loaded by the system administrator or by automatic boot-time initialization scripts.

Below we present a high level view of the GINSU system architecture.

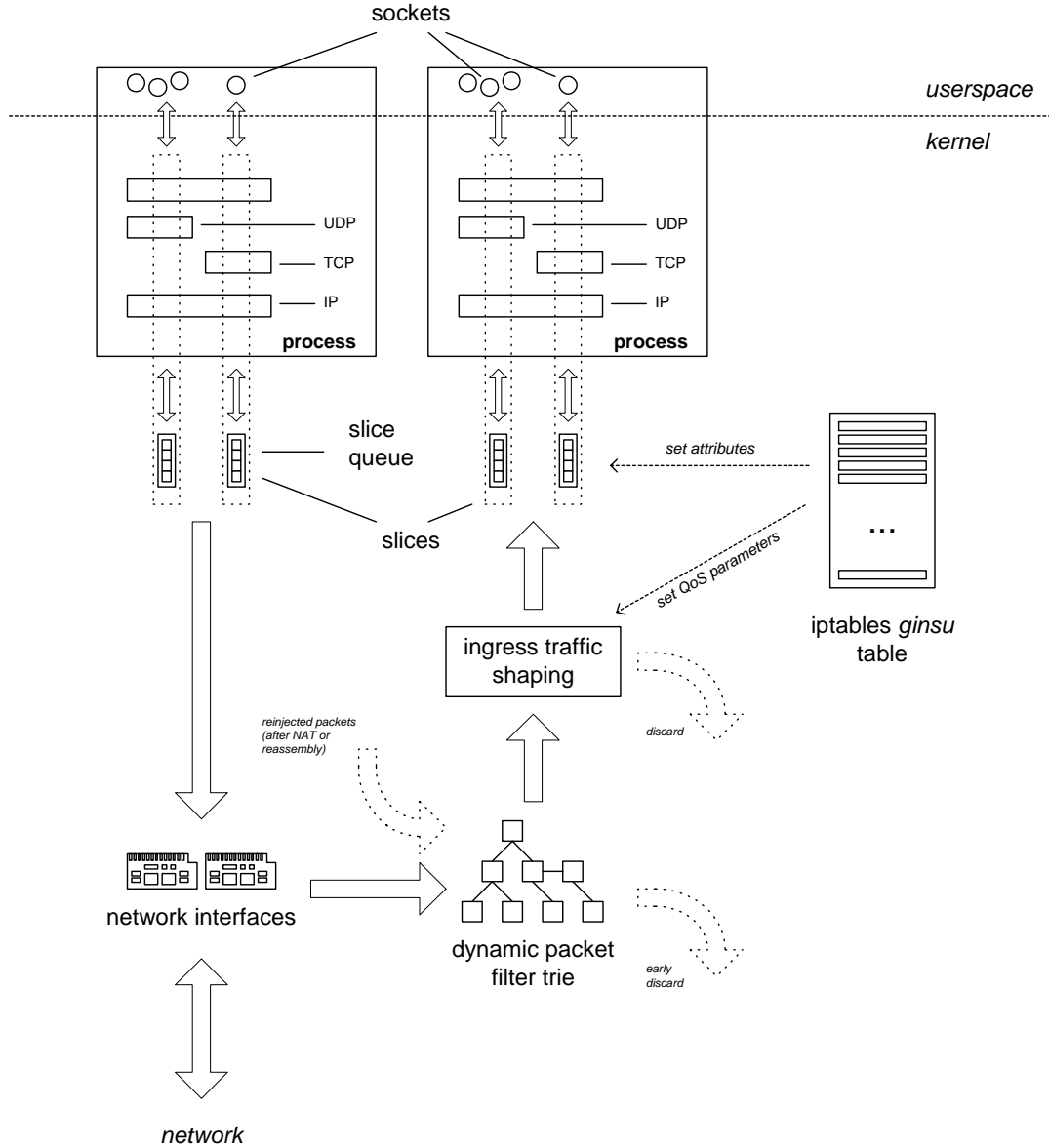


Figure 1: The High-level GINSU System Architecture

The wide arrows in the above diagram represent the flow of network packets through a GINSU-enhanced kernel. The diagram is ordered from bottom to top from lowest- (i.e. closest to hardware) level up through successive layers of abstraction to high- (i.e. application) level network processing functions.

GINSU builds upon the best ideas from Scout/Escort (the *path* abstraction), SILK (open source host integration), Exokernel (logical stack encapsulation and dynamic packet classification), and NAI Labs' AMP Channel Stack (traffic isolation) and combines and extends them into a portable, maintainable, and manageable host package. We embrace a kernel-space approach rather than a user-space approach, as research results from the latter have tended to exhibit unacceptable performance and generally require kernel-space

modifications anyway. The cornerstone of the GINSU design is the concept of traffic slices. A *slice* subdivides network traffic into demultiplexed "streams." These slices are independently scheduled and monitored. One or more catch-all slices are allocated for unauthenticated traffic. At the heart of GINSU is the ability to track per-traffic-slice resource consumption in order to provide fairness between authenticated and unauthenticated traffic resource use, as well as provide non-interference between individual traffic slices. Slices may be allocated for each endpoint created by an application or kernel process for receiving or transmitting network traffic, or for some aggregate, for example, a particular destination TCP port on the host, or a particular source subnet. The slice hierarchy and the assignment of traffic classes to slices is determined by the administrator and performed at run-time through the use of command-line utilities by the administrator. As traffic is partitioned into slices, the administrator may also set limits on or pre-allocate reservations of network-related kernel resources. These include: connection tracking structures; message buffers; available bandwidth; and available CPU processing time.

At the earliest possible point in per-packet processing, GINSU determines, based on endpoint addressing information contained within the packet, whether the packet belongs either to 1) a known and authenticated receiver, 2) an unauthenticated receiver (either new or unknown), or 3) no receiver at all. This process is known as *early demux*. (Note that the term "receiver" has multiple connotations here: by "receiver" we mean both a specific endpoint and also the traffic slice containing this endpoint.) This analysis is performed asynchronously at interrupt time using an efficient tri-based approach pioneered in the Exokernel and ported to the Linux platform as part of this research. Tries are populated on socket (endpoint) creation and address binding, as appropriate, and depopulated upon socket destruction. Based on the results of this analysis, GINSU may simply discard the packet before any host resources or kernel packet processing time is consumed at all. Otherwise, the packet is either marked for traffic shaping, or it is queued for subsequent standard protocol processing. Unless the received packet is destined for an endpoint created by the currently executing process, it will be deferred until that process is selected for execution by the OS scheduler. This feature is known as *lazy receiver processing*. Aside from early demux, GINSU may also be distinguished from standard Linux here by its use of lazy receiver processing. The combination of early discard and lazy receiver processing implies that, under certain circumstances, especially those likely to be encountered during an attempted denial-of-service attack, GINSU requires *less* processing than unmodified Linux.

If a packet is marked for traffic-shaping then further GINSU processing takes place immediately. GINSU implements a modular framework for ingress traffic-shaping modeled after existing Linux facilities for egress traffic-shaping. Within this framework we modified the existing Linux hierarchical token bucket (HTB) egress packet scheduler for ingress operation.

The GINSU software is partitioned into four loadable kernel modules. The *ginsu_common* module contains generic functions for intercepting kernel actions (the hook API), a very useful and flexible hash table implementation (the map API), and a

generic hierarchical resource management framework (the resource API). The *ginsu_low* module implements all intelligent GINSU functionality, utilizing the hook and resource APIs provided by *ginsu_common*. This functionality includes the dynamic packet filtering facility (or DPF), the ingress traffic-shaping framework (the ingress-shaping API), a command-line-based management interface, and all slice and socket resource tracking logic. The *ginsu_sch_htb* module implements a hierarchical token bucket (or HTB) traffic-shaping algorithm for use with *ginsu_low*'s ingress-shaping API. Finally, the *ginsu_proc* module provides a read-only view into internal GINSU state via a file tree in the Linux `/proc` filesystem.

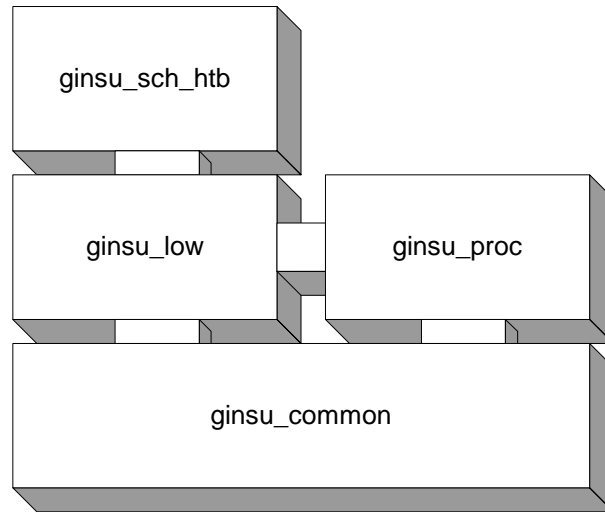


Figure 2: Module Stacking in the GINSU System

Developers wishing to extend the GINSU framework to allow additional resource accounting or to provide some other enhanced functionality should make themselves familiar with the GINSU source code. Significant aspects of that source code are described below. Administrators interested in how the GINSU system operates, from a high-level perspective, may refer to document TR-XXX “GINSU Administrative Guide” and the pecially marked sections in this document.

2.1 *ginsu_common*

The *ginsu_common* module contains generic functions for intercepting kernel actions (the hook API), a very useful and flexible hash table implementation (the map API), and a generic hierarchical resource management framework (the resource API).

2.1.1 *Hooks*

GINSU hooks provide additional generic interception points for various actions above and beyond those provided by a standard Linux kernel. Also, various hooks for internal GINSU actions are provided. Currently the following hook points are defined:

GINSU_HOOK_SCHEDULER

within the system scheduler, invoked whenever a context switch occurs

GINSU_HOOK_TIMER

within the system clock (timer) soft interrupt handler, invoked once per time-slice (100 HZ default on the Intel IA-32 platform)

GINSU_HOOK_SYS_PRE_FORK

invoked at the start of the fork syscall, just prior to execution of privileged (kernel) code

GINSU_HOOK_SYS_POST_FORK

invoked from within the fork syscall, subsequent to execution of privileged code

GINSU_HOOK_SYS_PRE_ACCEPT

invoked at the start of the accept socket syscall, just prior to execution of privileged code

GINSU_HOOK_SYS_POST_ACCEPT

invoked from within the accept socket syscall, subsequent to execution of privileged code

GINSU_HOOK_SYS_PRE_BIND

invoked at the start of the bind socket syscall, just prior to execution of privileged code

GINSU_HOOK_SYS_POST_BIND

invoked from within the bind socket syscall, subsequent to execution of privileged code

GINSU_HOOK_SYS_PRE_CLOSE

invoked at the start of the close syscall, just prior to execution of privileged code

GINSU_HOOK_SYS_POST_CLOSE

invoked from within the close syscall, subsequent to execution of privileged code

GINSU_HOOK_SYS_PRE_CONNECT

invoked at the start of the connect socket syscall, just prior to execution of privileged code

GINSU_HOOK_SYS_POST_CONNECT

invoked from within the connect socket syscall, subsequent to execution of privileged code

GINSU_HOOK_SYS_PRE_LISTEN

invoked at the start of the listen socket syscall, just prior to execution of privileged code

`GINSU_HOOK_SYS_POST_LISTEN`

invoked from within the listen socket syscall, subsequent to execution of privileged code

`GINSU_HOOK_SYS_PRE_SHUTDOWN`

invoked at the start of the shutdown socket syscall, just prior to execution of privileged code

`GINSU_HOOK_SYS_POST_SHUTDOWN`

invoked from within the shutdown socket syscall, subsequent to execution of privileged code

`GINSU_HOOK_SYS_PRE_SOCKET`

invoked at the start of the 'socket' socket syscall, just prior to execution of privileged code

`GINSU_HOOK_SYS_POST_SOCKET`

invoked from within the 'socket' socket syscall, subsequent to execution of privileged code

`GINSU_HOOK_SYS_PRE_SOCKETPAIR`

invoked at the start of the `socketpair` socket syscall, just prior to execution of privileged code

`GINSU_HOOK_SYS_POST_SOCKETPAIR`

invoked from within the `socketpair` socket syscall, subsequent to execution of privileged code

`GINSU_HOOK_SYS_PRE_EXIT`

invoked at the start of the exit syscall, just prior to execution of privileged code (control will not return to the application after privileged actions complete)

`GINSU_HOOK_SYS_POST_EXIT`

invoked from within the exit syscall, subsequent to execution of privileged code (control will not return to the application after privileged actions complete)

`GINSU_HOOK_PKT_RX`
 invoked from the network soft interrupt whenever a packet is received from a network device

`GINSU_HOOK_PKT_TX`
 invoked whenever a packet is presented to a network device for immediate transmission

`GINSU_HOOK_SLICE_CREATE`
 invoked by `ginsu_low` whenever a GINSU slice is created

`GINSU_HOOK_SLICE_DESTROY`
 invoked by `ginsu_low` whenever a GINSU slice is destroyed

`GINSU_HOOK_SLICE SOCK_ASSOCIATE`
 invoked by `ginsu_low` whenever a GINSU socket resource is associated with a parent slice resource

`GINSU_HOOK_SLICE SOCK_DISASSOCIATE`
 invoked by `ginsu_low` whenever a GINSU socket resource is disassociated from a parent slice resource

`GINSU_HOOK_SLICE_TASK_ASSOCIATE`
 invoked by `ginsu_low` whenever a GINSU socket resource is associated with a parent task resource

`GINSU_HOOK_DPF_INSERT`
 invoked by `ginsu_low` whenever a dynamic packet filter rule is about to be inserted into the system trie

`GINSU_HOOK_DPF_DELETE`
 invoked by `ginsu_low` whenever a DPF filter rule is about to be removed from the system trie

`GINSU_HOOK_GET_PROC_STATS`
 invoked by `ginsu_proc` to collect a human-readable list of global statistics and indicators for presentation to the human user via the `/proc` interface

A user of the hook API first registers one or more functions to be called at a specific hook site. These functions may be either passive, meaning they will not seek to alter decisions made by the kernel, or authoritative, meaning they may seek to alter decisions made by the kernel. Authoritative hooks are run before passive hooks. In the event that an authoritative hook cancels the current action, no further authoritative or any passive hooks will be invoked and an error condition will be signaled to the kernel. At any point a hook function may be unregistered.

```
int ginsu_hook_register (int where, int type, ginsu_hook_func_t f);
int ginsu_hook_unregister (int where, ginsu_hook_func_t f);
```

Internally, the hook API provides functions for use within GINSU kernel modules for driving hook invocations.

```
int ginsu_hook_call (int where, ...);
```

The exact number and type of arguments passed to the hook function vary from hook site to hook site, so variadic functions are used.

2.1.2 Maps

The ginsu_common module also provides a generic hash table implementation derived from the Scout [?] operating system. Using this "map" interface, a GINSU programmer may uniformly manage dynamic keyed lookup of a collection of data items of arbitrary size.

These functions create and destroy map structures, respectively:

```
ginsu_map_t ginsu_map_create (int nbuckets, int key_size);  
void        ginsu_map_destroy (ginsu_map_t m);
```

This function returns the current number of items contained within a map:

```
int          ginsu_map_count (ginsu_map_t m);
```

This function returns the size in octets of fixed-sized keys or -1 for variable (string) keys.

```
size_t       ginsu_map_key_size (ginsu_map_t m);
```

These functions manage the insertion and removal of items with fixed size keys into/from a map:

```
ginsu_map_binding_t ginsu_map_bind (ginsu_map_t m, const void * key,  
                                     unsigned long value);  
int                 ginsu_map_unbind (ginsu_map_t m, const void * key);  
int                 ginsu_map_remove_binding (ginsu_map_t m,  
                                               ginsu_map_binding_t b);
```

This function performs a keyed lookup of an item within a map:

```
int ginsu_map_resolve (ginsu_map_t m, const void * key, int key_size,  
                      unsigned long * value);
```

These functions allow the user to enumerate over all items contained within a map:

```
void          ginsu_map_walk_init (ginsu_map_walk_t, ginsu_map_t m);  
ginsu_map_el_t ginsu_map_walk_next (ginsu_map_walk_t w);  
void          ginsu_map_walk_done (ginsu_map_walk_t w);
```

Finally, these functions manage insertion and removal of items with variable sized (string) keys into/from a map:

```

ginsu_map_binding_t ginsu_map_var_bind (ginsu_map_t m, const void * key,
                                         int key_size, unsigned long value);
int ginsu_map_var_unbind (ginsu_map_t m,
                          const void * key, int key_size);
int ginsu_map_var_resolve (ginsu_map_t m,
                           const void *key, int key_size,
                           unsigned long * value);

```

2.1.3 Resource Management

The `ginsu_common` module also exports the GINSU generic hierarchical resource management framework. Under this regime, arbitrarily sized blobs of binary data (presumably kernel resources or internal GINSU state) may be uniformly managed. Though the framework is agnostic about the internal structure of resources it does require that each distinct resource type be assigned a unique integer selector. One or more attributes may be associated with a given resource. The framework is also agnostic regarding the internal structure of attributes. However, for each resource type, unique integer selectors must be used when getting or setting an attribute. Relationships among resources are maintained in a directed graph. Parent-child relationships are represented as bidirectional edges within that graph. When each resource is created, type-specific *manage* and *release* methods are provided by the caller, for initializing and for cleaning up state, respectively. The primary benefit of the GINSU resource management framework is the automatic destruction (release) of child resources when parents are explicitly destroyed. This framework also makes trivial resource enumeration up from children to parents, or down from parents to children. Separately, per-type maps are used to store pointers to every instance of a given resource type. Type maps are primarily used for efficient enumeration of all instances of a particular type, but because type map entries are indexed by unique type-specific keys, a programmer may exploit these maps for rapid location of a particular resource instance without resorting to an expensive traversal of the resource hierarchy.

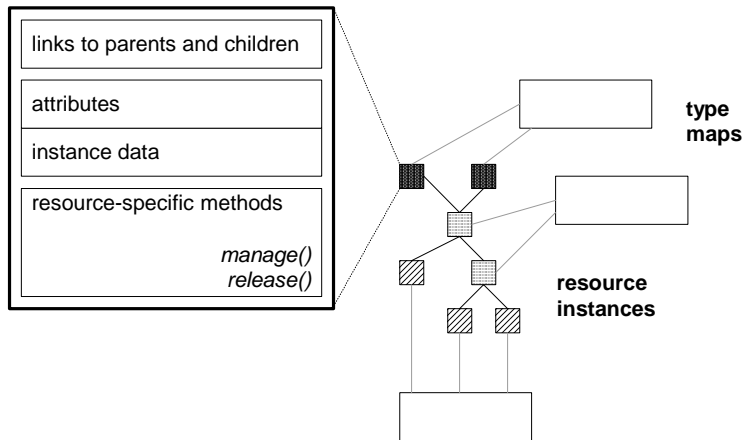


Figure 3: The GINSU Generic Resource Framework

This function creates a new resource and associates it with a unique key, a type-specific manage function, and a type-specific release function:

```
ginsu_resource_t ginsu_resource_new (int type, void * key,
                                     int key_len, void * data, int data_len,
                                     int (*manage)(ginsu_resource_t),
                                     int (*release)(ginsu_resource_t));
```

This function frees (releases) a resource and any children:

```
void ginsu_resource_free (ginsu_resource_t r);
```

This function frees all resources of a given type and any of their children, regardless of type:

```
void ginsu_resource_free_all (int type);
```

These functions allow the user to get or set resource attributes:

```
int ginsu_resource_get_attr (ginsu_resource_t r, int attr,
                             void ** data, int * data_len);
int ginsu_resource_set_attr (ginsu_resource_t r, int attr,
                             void * data, int data_len);
int ginsu_resource_get_attr_simple (ginsu_resource_t r, int attr,
                                     unsigned long * val);
int ginsu_resource_set_attr_simple (ginsu_resource_t r, int attr,
                                     unsigned long val);
void * ginsu_resource_get_data (ginsu_resource_t r, int * data_len);
int ginsu_resource_set_data (ginsu_resource_t r, void * data,
                             int data_len);
```

These functions enumerate the resource hierarchy starting at a specific resource if 'r' is non-NULL, or starting at the roots of the specified type otherwise. During the enumeration, canned actions are taken if any of the desired resource type is found.

```
int ginsu_resource_enum_attr_find_first_up (ginsu_resource_t r,
                                           int which, int attr, ginsu_resource_t * result);
int ginsu_resource_enum_attr_find_first_down (ginsu_resource_t,
                                              int which, int attr, ginsu_resource_t * result);
int ginsu_resource_enum_attr_simple_min_up (ginsu_resource_t r,
                                           int which, int nr_attrs, int * attrs,
                                           unsigned long * vals);
int ginsu_resource_enum_attr_simple_min_down (ginsu_resource_t,
                                              int which, int nr_attrs, int * attrs,
                                              unsigned long * vals);
int ginsu_resource_enum_attr_simple_max_up (ginsu_resource_t r,
                                           int which, int nr_attrs, int * attrs,
                                           unsigned long * vals);
int ginsu_resource_enum_attr_simple_max_down (ginsu_resource_t,
                                              int which, int nr_attrs, int * attrs,
                                              unsigned long * vals);
int ginsu_resource_enum_attr_simple_sum_up (ginsu_resource_t r,
                                           int which, int nr_attrs, int * attrs,
                                           unsigned long * vals);
int ginsu_resource_enum_attr_simple_sum_down (ginsu_resource_t,
                                              int which, int nr_attrs, int * attrs,
                                              unsigned long * vals);
int ginsu_resource_enum_attr_simple_inc_up (ginsu_resource_t r,
                                           int which, int nr_attrs, int * attrs,
                                           unsigned long * vals);
int ginsu_resource_enum_attr_simple_inc_down (ginsu_resource_t,
                                              int which, int nr_attrs, int * attrs,
                                              unsigned long * vals);
int ginsu_resource_enum_attr_simple_dec_up (ginsu_resource_t r,
                                           int which, int nr_attrs, int * attrs,
                                           unsigned long * vals);
int ginsu_resource_enum_attr_simple_dec_down (ginsu_resource_t,
                                              int which, int nr_attrs, int * attrs,
                                              unsigned long * vals);
```

These functions will enumerate the resource hierarchy and compare two attributes: a limit, and a count. If the count attribute for a resource is found to exceed the limit attribute for that same resource, a pointer to that resource will be returned in *result*.

```
int ginsu_resource_enum_attr_simple_test_limit_up (ginsu_resource_t r,
                                                  int which, int limit_attr, int count_attr,
                                                  ginsu_resource_t * result);
int ginsu_resource_enum_attr_simple_test_limit_down (ginsu_resource_t r,
                                                    int which, int limit_attr, int count_attr,
                                                    ginsu_resource_t * result);
```

This function will allow the user to individually enumerate all resources of a given type:

```
ginsu_resource_t ginsu_resource_enumerate (int type,
                                           unsigned long * cookie);
```

This function locates a resource given a type and a unique key:

```
ginsu_resource_t ginsu_resource_find (int type, void * key, int key_len);
```

These functions let the user manage parent-child relationships among resources:

```
int ginsu_resource_parent (ginsu_resource_t r, ginsu_resource_t parent);
int ginsu_resource_reparent (ginsu_resource_t r,
                             ginsu_resource_t old_parent, ginsu_resource_t new_parent);
int ginsu_resource_unparent (ginsu_resource_t r, ginsu_resource_t parent);
```

2.1.4 Logging

When resource limits are exceeded, or when current system conditions require proactive action to maintain a specified resource reservation, GINSU will send a detailed message to the system logging (syslog) facility. A table describing the format of these messages and presenting an example follows:

<i>timestamp</i>	<i>module</i>	<i>facility</i>	<i>resource</i>	<i>Message data</i>
Jun 30 14:21:26	GINSU:	SLICE:	socket_limit	Sid=31 now=100 limit=100

The following function provides the programmatic interface to this facility:

```
#define GINSU_SYSLOG_EMERG      0      system is unusable
#define GINSU_SYSLOG_ALERT     1      action must be taken immediately
#define GINSU_SYSLOG_CRIT      2      critical conditions
#define GINSU_SYSLOG_ERR       3      error conditions
#define GINSU_SYSLOG_WARNING   4      warning conditions
#define GINSU_SYSLOG_NOTICE    5      normal, but significant conditions
#define GINSU_SYSLOG_INFO      6      informational
#define GINSU_SYSLOG_DEBUG     7      debug level messages

int ginsu_syslog (int severity, char * module, char * facility,
                  char * resource, char * fmt, ...);
```

2.2 ginsu_low

The ginsu_low module implements all intelligent GINSU functionality, utilizing the hook and resource APIs provided by ginsu_common. This functionality includes the dynamic packet-filtering facility (or DPF), the ingress traffic-shaping framework (the ingress-shaping API), a command-line-based management interface, and all slice and socket resource tracking logic.

As previously discussed, GINSU partitions network traffic into distinct slices. Each slice may be individually associated with resource limits or reservations. A default slice collects all traffic not otherwise directed. A hierarchical token bucket traffic-shaping scheme (implemented in conjunction with the ginsu_sch_htb module) provides effective limits and reservations on different classes of network traffic. Simple limits on message buffer memory, connection table entries, and CPU timeslice consumption are also

provided. Simple reservations for connection table entries and network bandwidth usage are supported. Internally, Linux creates a socket for each distinct endpoint. GINSU dynamically creates socket resources as Linux sockets are created or destroyed by applications. Sockets are automatically assigned to slices according to their source and/or destination endpoint addressing information. ginsu_low uses the socket syscall interposition points provided by ginsu_common to monitor application activity for socket creation, endpoint address binding, and socket destruction events.

2.2.1 Installing Resource Constraints

An administrator may install slice partitioning rules and their corresponding reservations and limits through extensions to the standard Linux *IPTables* packet-matching rule language. IPTables is the Linux interface to the network subsystem for use in firewalling, packet inspection, or packet rewriting applications. We have added a custom GINSU iptables table and LIMIT and RESERVE rule targets.

```
Usage: iptables -[ADC] chain rule-specification [options]
       iptables -[RI] chain rulenum rule-specification [options]
       iptables -D chain rulenum [options]
       iptables -[LFZ] [chain] [options]
       iptables -[NX] chain
       iptables -E old-chain-name new-chain-name
       iptables -P chain target [options]
```

LIMIT target options:

--limit-timeslice <percent>	Limit total percentage of each timeslice that may be consumed from the owner process for this flow.
--limit-bandwidth <bits>	Limit flow bandwidth to <bits> bits/second.
--limit-bandwidth-octets <octets>	Like --limit-bandwidth, but in units of octets instead of bits.
--limit-sockets <count>	Cap total number of unique endpoints allowed for the given flow.
--limit-connections <count>	Cap total number of connections for the given flow (includes half-open).
--limit-queue <count>	Limit the maximum number of queued sk_buffs for the given flow.
--renice <priority>	Renices (decreases) base priority of owner process to the given limit.
--euid <euid>	Only match if effective user == euid.
--egid <egid>	Only match if effective group == egid.
--sid <sid>	Attach rule to slice with given SID.

RESERVE target options:

--reserve-bandwidth <bits>	Reserve flow bandwidth of <bits> bits/second.
--reserve-bandwidth-octets <bytes>	Like --reserve-bandwidth, but in units of octets instead of bits.
--reserve-connections <count>	Reserve a number of connection slots for the given flow.
--renice <priority>	Renices (increases) base priority of owner process up to the given reservation.
--euid <euid>	Only match if effective user == euid.
--egid <egid>	Only match if effective group == egid.
--sid <sid>	Attach rule to slice with given SID.

These custom targets are in addition to the traditional iptables matches and extensions, portions of which are excerpted below from the iptables manual:

PARAMETERS

The following parameters make up a rule specification (as used in the add, delete, insert, replace and append commands).

-p, --protocol [!] *protocol*

The protocol of the rule or of the packet to check. The specified protocol can be one of *tcp*, *udp*, *icmp*, or *all*, or it can be a numeric value, representing one of these protocols or a different one. A protocol name from */etc/protocols* is also allowed. A "!" argument before the protocol inverts the test. The number zero is equivalent to *all*. Protocol *all* will match with all protocols and is taken as default when this option is omitted.

-s, --source [!] *address[/mask]*

Source specification. *Address* can be either a hostname, a network name, or a plain IP address. The *mask* can be either a network mask or a plain number, specifying the number of 1's at the left side of the network mask. Thus, a mask of 24 is equivalent to 255.255.255.0. A "!" argument before the address specification inverts the sense of the address. The flag **--src** is a convenient alias for this option.

-d, --destination [!] *address[/mask]*

Destination specification. See the description of the **-s** (source) flag for a detailed description of the syntax. The flag **--dst** is an alias for this option.

-i, --in-interface [!] [*name*]

Optional name of an interface via which a packet is received (for packets entering the **INPUT**, **FORWARD** and **PREROUTING** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, the string "+" is assumed, which will match with any interface name.

-o, --out-interface [!] [*name*]

Optional name of an interface via which a packet is going to be sent (for packets entering the **FORWARD**, **OUTPUT** and **POSTROUTING** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, the string "+" is assumed, which will match with any interface name.

MATCH-EXTENSIONS

iptables can use extended packet-matching modules. These are loaded in two ways: implicitly, when **-p** or **--protocol** is specified, or with the **-m** or **--match** options, followed by the matching module name; after these, various extra command line options become available, depending on the specific module. You can specify multiple extended

match modules in one line, and you can use the **-h** or **--help** options after the module has been specified to receive help specific to that module.

The following are included in the base package, and most of these can be preceded by a **!** to invert the sense of the match.

Tcp

These extensions are loaded if `--protocol tcp` is specified. It provides the following options:

--source-port [!] [*port[:port]*]

Source port or port range specification. This can either be a service name or a port number. An inclusive range can also be specified, using the format *port:port*. If the first port is omitted, "0" is assumed; if the last is omitted, "65535" is assumed. If the second port is greater than the first they will be swapped. The flag **--sport** is an alias for this option.

--destination-port [!] [*port[:port]*]

Destination port or port range specification. The flag **--dport** is an alias for this option.

--tcp-flags [!] *mask comp*

Match when the TCP flags are as specified. The first argument is the flags which we should examine, written as a comma-separated list, and the second argument is a comma-separated list of flags which must be set. Flags are: **SYN ACK FIN RST URG PSH ALL NONE**. Hence the command

`iptables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST SYN`
will only match packets with the SYN flag set, and the ACK, FIN and RST flags unset.

[!] **--syn**

Only match TCP packets with the SYN bit set and the ACK and FIN bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will prevent incoming TCP connections, but outgoing TCP connections will be unaffected. It is equivalent to **--tcp-flags SYN,RST,ACK SYN**. If the **!** flag precedes the **--syn**, the sense of the option is inverted.

--tcp-option [!] *number*

Match if TCP option set.

Udp

These extensions are loaded if `--protocol udp` is specified. It provides the following options:

--source-port [!] [*port[:port]*]

Source port or port range specification. See the description of the **--source-port** option of the TCP extension for details.

--destination-port [!] [*port[:port]*]

Destination port or port range specification. See the description of the **--destination-port** option of the TCP extension for details.

For example, the following rule partitions traffic directed at a local Web server into a distinct slice with a 60 mega-bit-per-second upper bound on available network bandwidth. (Leftover bandwidth is then available for urgent or administrative actions.)

```
iptables -t ginsu -A PREROUTING -p tcp --destination-port http
        -j LIMIT --limit-bandwidth 62914560
```

The ‘limit-queue’ limit option sets a maximum bound on the length of the slice queue. This is not an aggregate limit, however – each child slice gets the same limit. Setting the queue limit to zero effectively discards matching traffic at DPF demultiplexing time, which is the most effective way to shed unwanted traffic. Setting the bandwidth limit of a slice to zero will also drop traffic, but not as efficiently. One can install DPF filters for obvious attack traffic with “null route” straight to early discard as follows:

```
iptables -t ginsu -A PREROUTING -s netblock/mask --in-interface iface
        -j LIMIT --limit-queue 0
```

The ‘renice’ limit and reservation options will dynamically lower or raise, respectively, the base system scheduling priority of the owner process for the slice. “Reserving” a ‘renice’ reservation will raise the process priority if its priority after slice association is too low. Conversely, setting a ‘renice’ limit will lower the process priority if its priority after slice association is too high. Note that the POSIX limit for scheduling priority range is from –20 (lowest) to 19 (highest).

Important Note: Slice partitioning rules are applied only when new slices are created. Ideally, newly installed limits and reservations should be retroactively applied to existing slices. However, our prototype does not do this. Accordingly, a GINSU administrator should define slice/traffic sorting rules early, before any such traffic is processed by the host; or, if protective traffic limits have been installed, arrange for the effective service to restart any existing connections.

When the `ginsu_low` module is first loaded, it waits for additional signals from the user before commencing full operation. The GINSU administrator may separately enable or

disable slice sorting, lazy receiver processing, or ingress traffic-shaping by setting a special GINSU-specific socket option using the standard UNIX `setsockopt()` function. These functions may be enabled or disabled at any time. The following snippet of C code demonstrates how this is done:

```
#define GINSU_SIOCTL          40
#define GINSU_SIOCTL_START    1
#define GINSU_SIOCTL_STOP     2
#define GINSU_SUBSYS_SLICE    1
#define GINSU_SUBSYS_SHAPING  2
#define GINSU_SUBSYS_LRP      3

int fd;
struct { int command; int what; } args;

/* get a socket */
fd = socket(PF_INET, SOCK_DGRAM, PF_UNSPEC);

/* start slice sorting */
args.command = GINSU_SIOCTL_START;
args.what = GINSU_SUBSYS_SLICE;
setsockopt(fd, SOL_IP, GINSU_SIOCTL, &args);

/* start lazy receiver processing */
args.command = GINSU_SIOCTL_START;
args.what = GINSU_SUBSYS_LRP;
setsockopt(fd, SOL_IP, GINSU_SIOCTL, &args);

/* stop lazy receiver processing */
args.command = GINSU_SIOCTL_STOP;
args.what = GINSU_SUBSYS_LRP;
setsockopt(fd, SOL_IP, GINSU_SIOCTL, &args);

/* when done, close (release) the socket */
close(fd);
```

The GINSU source distribution includes simple utilities written in Perl for managing this process from the command line. The ‘up’ and ‘down’ utilities can be used once the GINSU modules have been loaded as follows:

```
up: usage: up <subsystem1> [...<subsystemN>]
      where <subsystem> is one of SLICE, LRP, or SHAPING

down: usage: down <subsystem1> [...<subsystemN>]
      where <subsystem> is one of SLICE, LRP, or SHAPING
```

Note that once slice-sorting has commenced, it must be stopped before the `ginsu_low` module may be unloaded with the Linux ‘`rmmod`’ (*remove module*) loadable kernel module management utility.

2.2.2 Startup Sequence

Once the `ginsu_low` module is initialized it immediately begins intercepting packets as they arrive from the network. This is done using a hook function registered with the Linux ‘netfilter’ packet interception facility on the `NF_IP_PRE_ROUTING` chain. From within this hook function GINSU gains control upon the reception of a packet and may modify, steal, or do nothing with the packet, before returning control to the Linux kernel. Within GINSU, each packet is first processed by the dynamic packet filter (DPF) facility, which attempts to match the contents of the packet to a trie of {offset,length,mask,value} tuples. These tuples are installed automatically by `ginsu_low` as sockets are created and bound to transport endpoints by applications. A DPF lookup operation returns either a slice pointer or the root, or default, slice pointer if an explicit listener was not found. The default slice operates with low priority. This arrangement automatically prioritizes expected traffic over unexpected, potentially unauthorized (or attack) traffic.

2.3 The Dynamic Packet Filter (DPF) API

The DPF trie is managed by the following functions:

```
void ginsu_dpf_begin (struct dpf_ir ** ir);
void ginsu_dpf_end (struct dpf_ir ** ir);
int ginsu_dpf_insert (struct ginsu_sock * ss, struct dpf_ir * ir);
int ginsu_dpf_delete (struct ginsu_sock * ss);
void ginsu_dpf_printir (char * buf, struct dpf_ir * ir);
int ginsu_dpf_atoms (struct dpf_ir * ir);
```

DPF rules are constructed out of one or more atoms specified with these functions:

Filter creation routines. *nbits* corresponds to 8, 16, 32 -- depending on the operation. `msg[byte_offset:nbits]` means to load *nbits* of the message at *byte_offset*.

Compare message value to constant:

```
msg[byte_offset:nbits] & mask == val

void ginsu_dpf_meq8 (struct dpf_ir * ir, u_int16_t byte_offset,
                    u_int8_t mask, u_int8_t val);
void ginsu_dpf_meq16 (struct dpf_ir * ir, u_int16_t byte_offset,
                     u_int16_t mask, u_int16_t val);
void ginsu_dpf_meq32 (struct dpf_ir * ir, u_int16_t byte_offset,
                     u_int32_t mask, u_int32_t val);
```

Compare message value to constant:

```
msg[byte_offset:nbits] & mask != val

void ginsu_dpf_not_meq8 (struct dpf_ir * ir, u_int16_t byte_offset,
                        u_int8_t mask, u_int8_t val);
void ginsu_dpf_not_meq16 (struct dpf_ir * ir, u_int16_t byte_offset,
                          u_int16_t mask, u_int16_t val);
```

```
void ginsu_dpf_not_meq32 (struct dpf_ir * ir, u_int16_t byte_offset,
                        u_int32_t mask, u_int32_t val);
```

Shift the base message pointer:

```
msg += (msg[byte_offset:nbits] & mask) << shift;

void ginsu_dpf_mshift8 (struct dpf_ir * ir, u_int16_t offset,
                      u_int8_t mask, u_int8_t shift);
void ginsu_dpf_mshift16 (struct dpf_ir * ir, u_int16_t offset,
                       u_int16_t mask, u_int8_t shift);
void ginsu_dpf_mshift32 (struct dpf_ir * ir, u_int16_t offset,
                       u_int32_t mask, u_int8_t shift);
```

Shift the base message pointer by a constant:

```
msg += nbytes.

void ginsu_dpf_shifti (struct dpf_ir * ir, u_int16_t nbytes);
```

The GINSU DPF facility is a port of the dynamic packet-filtering subsystem of the MIT Exokernel. Accordingly it has its strengths – simplicity and efficiency – and its weaknesses – lack of support for packet headers of variable size, or for fragmented IP packets. We have partially implemented a countermeasure for the latter shortcoming: a special GINSU kernel process that will consume fragmented IP packets, reassemble them, and then either retry the DPF process once all fragments have been received, or discard partially reassembled packets if not all fragments are received within a few seconds. However, this code remains untested and probably will not function correctly without minor bug fixes. Regarding the former shortcoming, the AMP project at Network Associates Laboratories encountered this limitation and addressed it by augmenting DPF with a special operator that would shift the base message pointer based on special knowledge of the IPv4 and TCP header structures. In this way, filters could be specified as if the variable portions of those headers simply did not exist. A refined GINSU prototype could implement similar functionality.

2.4 Queue Management

Once traffic is sorted it is then categorized for traffic-shaping according to any bandwidth limits or reservations associated with the matching slice. Traffic-shaping is performed by a generic framework that can support multiple arbitrary queuing disciplines. Currently, we provide a hierarchical token bucket queuing discipline, implemented in the separate `ginsu_sch_htb` module, and a simple rate-limiting scheme implemented within `ginsu_low` that does not depend on any external module. Under normal operation, the `ginsu_sch_htb` module is automatically loaded and configured when bandwidth limits or reservations are set via the GINSU iptables interface.

Ingress packet-queuing disciplines are managed with these functions:

```
int ginsu_qdisc_register (struct ginsu_qdisc_ops *);
int ginsu_qdisc_unregister (struct ginsu_qdisc_ops *);
```

Shared rate tables are managed with these functions:

```
struct ginsu_qdisc_rtab * ginsu_qdisc_get_rtab (
                                struct ginsu_qdisc_ratespec *,
                                u_int32_t *);
void ginsu_qdisc_put_rtab (struct ginsu_qdisc_rtab *);
```

Internally, queuing disciplines and their associated traffic classes are instantiated, destroyed, and modified with these functions:

```
struct ginsu_qdisc * ginsu_qdisc_create (char *, u_int32_t,
                                         unsigned long *);
struct ginsu_qdisc * ginsu_qdisc_find (struct net_device *,
                                         u_int32_t);
int ginsu_qdisc_destroy (struct ginsu_qdisc *);
int ginsu_qdisc_graft (struct net_device *, struct ginsu_qdisc *,
                      u_int32_t, struct ginsu_qdisc *, struct ginsu_qdisc **);
int ginsu_qdisc_change_class (struct ginsu_qdisc *, u_int32_t,
                              u_int32_t, unsigned long *);
```

In cooperation with the `ginsu_sch_htb` module, `ginsu_low` provides hierarchical token bucket traffic-shaping capabilities by default. Information on the theory of operation of HTB is outside the scope of this document. More information may be found at the Linux HTB home page, at <http://luxik.cdi.cz/~devik/qos/htb/>.

The `ginsu_low` module is the primary resource monitor. Internally it uses three major structures to account for network stack and host OS resource usage: `ginsu_task`, `ginsu_slice`, and `ginsu_sock` objects.

2.5 Slice Scheduling

One `ginsu_task` object is maintained for every OS task (also known as a *process*). Using this object, GINSU manages a run queue for slices with pending work: recall the earlier discussion regarding lazy receiver processing. LRP defers packet-processing work when packets are received from the networks that are not destined for the currently executing process. Such packets are queued in the incoming queue of their target slice. This slice, in turn, is flagged as runnable and placed on the run queue of the task that owns the slice. At every context switch, GINSU gains control from within the system scheduler logic just prior to invocation of user-level code. (*In order to accommodate this control transfer, the stock Linux kernel scheduler must be modified with a small patch included in the GINSU source distribution. The kernel must then be recompiled and reinstalled.*) Here, pending slices are removed from the run queue within the corresponding `ginsu_task` object and their incoming packet queues are serviced. In order to prevent excessive network level work from consuming all of the time slice for newly running process, further LRP processing is deferred until the next available time slice if LRP processing consumes more than 60% of the current time slice. (This corresponds to a period of six milliseconds on unmodified Linux kernels for the Intel IA-32 architecture.) This is done to insure that the user level application code has a reasonable amount of CPU time in order to make

progress given the new input from the network. Currently, this limit is hard-coded, but may be manually adjusted and put into effect by recompiling and reinstalling the GINSU modules.

Every `ginsu_task` object also contains a reference to the *root slice* for the process. In order to simplify the slice hierarchy, every task is allocated a root slice, which is thereafter the initial and default owner for every slice subsequently created for traffic destined for that process. Thus, within the GINSU resource hierarchy, ultimately for every slice created a `ginsu_task` object is a parent object of that slice. When and if the task is destroyed by the operating system, as the task object is released, the resource framework will automatically and efficiently release any child `ginsu_slice` object, and any children of those slices, and so on, preserving endpoint tear down semantics upon application exit. The `ginsu_low` module, therefore, does not need to, and does not, explicitly manage this process.

```
struct ginsu_task
{
    unsigned long magic;
    spinlock_t lock;
    unsigned long flags;           /* operational mode */
    struct task_struct * task;     /* corresponding Linux task */
    /* slice run queue */
    TAILQ_HEAD(ginsu_slice) runq;  /* slices with work pending */
    TAILQ_HEAD(ginsu_slice) doneq; /* slices with work completed */
    struct ginsu_slice * root;     /* root slice for this task */
    ginsu_resource_t r;           /* pointer to resource for this task */
};
```

Figure 4: The GINSU Task Structure

2.5.1 Accounting Practices

For every traffic slice allocated for traffic-sorting or resource management bookkeeping, a `ginsu_slice` object is created and associated with an owner process. A locally unique integer identifier, termed a slice identifier (or SID), identifies every slice. There are also two work queues into which units of network protocol processing work may be placed to be serviced at a later time – one for packets received from the network, and one for packets scheduled by the owner process for transmission. The `ginsu_slice` object also contains state for use in limiting CPU and network bandwidth consumption in a hierarchical manner.

```

struct ginsu_slice
{
    unsigned long magic;
    spinlock_t lock;
    unsigned long flags;           /* operational mode */
    struct ginsu_task * owner;     /* owner GINSU task */
    int sid;                       /* slice identifier */
    /* work queues */
    struct {                       /* network work queues */
        struct sk_buff_head q;     /* sliceq[0] is RX queue */
        atomic_t count;           /* sliceq[1] is TX queue */
        atomic_t avail;
        ginsu_sliceq_func_t func;
    } sliceq[2];
    /* CPU limit bookkeeping */
    unsigned long ts_cap;          /* max timeslice portion allowed (1..100) */
    unsigned long ts_cur_cap;      /* currently consumed timeslice portion */
    volatile unsigned long * ts_cap_p; /* current CPU limit in force */
    volatile unsigned long * ts_cur_cap_p; /* current use for limit in force */
    /* linkage */
    TAILQ_ENTRY(ginsu_slice) runq_link;
    ginsu_resource_t r;            /* pointer to resource for this slice */
    void * root_dir;              /* procfs root dirent for this slice */
    void * sock_dir;              /* procfs socket dirent for this slice */
    /* simple (non-HTB) shaping bookkeeping */
    unsigned long max_rate;        /* max rate (per second) */
    unsigned long last;            /* rate for current 1-second interval */
};

```

Figure 5: The GINSU Slice Structure

For every high-level Linux socket created, a corresponding `ginsu_sock` object is created in order to store GINSU-specific per-socket state. This state includes: the identifier of any DPF filter inserted to direct packets to the appropriate active endpoint; the identifier of the ingress traffic-shaping class (if any) to which the endpoint's traffic will belong; and a list of all low-level Linux sockets created to service the endpoint. (There may be more than one low-level Linux socket created over the lifetime of the corresponding high-level Linux socket.) Every `ginsu_sock` object is associated with an owner slice. If the owning slice is destroyed, because of implicit actions taken by the resource management framework, all child `ginsu_sock` objects will be released as well. Also, during GINSU socket object creation, we overwrite a method pointer in the Linux kernel socket structure so we may receive notification when and if all low-level Linux sockets for the endpoint are destroyed. When this occurs, the corresponding GINSU state is released.

```

struct ginsu_sock
{
    unsigned long magic;
    spinlock_t lock;
    struct ginsu_slice * owner; /* owner GINSU slice */
    ginsu_resource_t r; /* pointer to resource for this socket */
    unsigned long mark; /* value to mark packets with for ingress QoS */
    int dpf_fid; /* DPF filter identifier */
    int nr_sk; /* count of associated low-level Linux sockets */
    LIST_HEAD(ginsu_sock_el) sk_list; /* list of associated low-level Linux sks */
    void * proc_data; /* DPF filter text for ginsu_proc */
    /* pointers to simple (non-HTB) shaping bookkeeping in force */
    volatile unsigned long * max_rate_p;
    volatile unsigned long * last_p;
};

```

Figure 6: The GINSU Sock Structure

2.5.2 Resource Constraint Enforcement

Resource usage is tracked in various ways that differ for the major resource classes. Flow bandwidth monitoring is implicit in the operation of the HTB component, so this state is distributed among the installed flow class configuration. GINSU also provides a very simple rate-limiting scheme that may be used in lieu of HTB queuing for LIMIT targets only. If this latter scheme is used, flow bandwidth monitoring is performed using state stored in `ginsu_slice` structures. The per-slice count of currently connected network flows is maintained in resource attributes within each slice resource. Likewise, slice resources are annotated with any connection limits installed by the administrator. At connection establishment time, within either the `connect()` or `accept()` syscalls, a breadth-first search of the slice resource hierarchy is performed. If the allowed connection count is exceeded, or a new connection would prevent a reservation from being serviced, the new connection will be rejected. CPU timeslice limits are maintained within the resource data for each slice. When child slices are associated with a slice with an active CPU limit, pointers within those children are updated to point to the appropriate fields in the parent. During LRP processing, these pointers are followed to insure that the aggregate processing time of all children of a CPU-limited slice do not exceed the limit in force. Per-slice socket buffer (SKB) limits are enforced whenever incoming work is to be posted to a work queue. If an in-force queue limit would be exceeded, the incoming packet is instead dropped. Likewise, if the size of the incoming packet, when added to the sum of the sizes of all queued packets, would exceed a limit on maximum allowed buffer memory for a slice, the incoming packet is also dropped.

When resource limits are exceeded or when resource utilization approaches a point where load must be shed in order to preserve a reservation, GINSU takes automatic action to enforce in-force limits and reservations. As these actions are undertaken, messages are sent to the system logging facility indicating the application(s), endpoint(s), and resource(s) identified as the cause of the out-of-line condition, what action was taken, and the result of that action.

2.5.2.1 Resource Constraint Violation Logging

When resource limits are exceeded, or when current system conditions require proactive action to maintain a specified resource reservation, GINSU will send a detailed message to the system logging (syslog) facility. A table describing the format of these messages and presenting an example follows:

<i>timestamp</i>	<i>module</i>	<i>facility</i>	<i>Resource</i>	<i>message data</i>
Jun 30 14:21:26	GINSU:	SLICE:	Socket limit	sid=31 now=100 limit=100

The following table enumerates all resources managed by ginsu_low for which messages may appear in the syslog upon an exception:

<i>facility</i>	<i>resource</i>	<i>Exception</i>
SLICE	sockets	Slice socket limit exceeded. “limit exceeded: sid=%d euid=%d egid=%d limit=%d”
	connections	Slice connection limit exceeded (includes half-open). “limit exceeded: sid=%d euid=%d egid=%d limit=%d”
	queue length	Slice queue entry count limit exceeded. Note: Messages for this exception will be rate-limited. “limit exceeded: sid=%d euid=%d egid=%d limit=%d”
TASK	timeslice	Timeslice limit in lazy receiver processing exceeded. “limit exceeded: pid=%d sid=%d euid=%d egid=%d limit=%d\%”
SHAPING	bandwidth	Moderate to high drop rate indicates class bandwidth limit exceeded. Note: Messages for this exception will be rate-limited. “limit exceeded: sid=%d euid=%d egid=%d limit=%d”

2.6 ginsu_proc

The ginsu_proc module provides a read-only view into internal GINSU state via a file tree in the Linux /proc filesystem.

```
/proc/net/ginsu/
|
+-- stat
|
+-- slice/
|
|   +-- <1st slice>
|   |
|   |   +-- stat
|   |   |
|   |   +-- sock/
|   |   |
|   |   |   +-- <1st socket>
|   |   |   |
|   |   |   |   ...
|   |   |   |   +-- <Nth socket>
|   |   |
|   |   ...
|   |   +-- <Nth slice>
```

The top-level, or global, ‘stat’ file provides a read-only listing of system-wide state, counters, and statistics. Various parts of GINSU register functions on the GINSU_HOOK_GET_PROC_STATS hook. When the user requests ‘stat’ this hook is invoked and the results are then passed to userspace. Below is an example listing from the global ‘stat’ file:

modeflags: SLICE LRP	<i>current GINSU operating mode</i>
slices: 13	<i>number of slices active in system</i>
sockets: 56	<i>number of sockets active in system</i>
slice_immed: 78644	<i>packets processed immediately (in context)</i>
slice_posted: 419857	<i>packets deferred into slice queues</i>
slice_lrp: 419857	<i>packets deferred for lazy receiver processing</i>
slice_dropped: 491	<i>packets dropped from slice queues</i>
slice_oom: 0	<i>packets dropped for insufficient buffer space</i>
cpu_overlimit: 0	<i>global count of timeslice over limit conditions</i>
queue_overlimit: 0	<i>global count of slice queue over limit conditions</i>
sock_overlimit: 0	<i>global count of socket alloc over limit conditions</i>
connect_overlimit: 0	<i>global count of connection over limit conditions</i>
dpfd_rx: 0	<i>packets received by defragmenter</i>
dpfd_inject: 0	<i>packets reinjected by defragmenter</i>
dpf_atoms: 59	<i>number of active DPF atoms</i>
dpf_atoms_highwater: 332	<i>max number of DPF atoms active at one time</i>
dpf_match_root: 2275	<i>packets matching trie root (default)</i>
dpf_match_leaf: 496221	<i>packets matching trie leaf</i>
pkt_rx: 498496	<i>packets received by host</i>
pkt_tx: 307652	<i>packets transmitted by host</i>

<code>pkt_should_lrp: 0</code>	<i>packets received with LRP disabled</i>
<code>pkt_marked: 193277</code>	<i>packets marked by ingress traffic-shaping</i>
<code>pkt_dropped: 1159</code>	<i>packets dropped by ingress traffic-shaping</i>
<code>pkt_stolen: 419857</code>	<i>packets “stolen” from Linux by GINSU</i>
<code>pkt_replaced: 4732</code>	<i>packets replaced by ingress traffic-shaping</i>
<code>pkt_bandwidth_overlimit: 0</code>	<i>global count of bandwidth over limit conditions</i>

The per-slice ‘stat’ file provides statistics and state information regarding a particular slice, as well as the values of any counts or limits that may be present as an attribute in the slice’s resource structure. Below is an example listing from a per-slice ‘stat’ file:

<code>/proc/net/ginsu/slice/7/stat:</code>	
<code>id: 7</code>	<i>slice identifier</i>
<code>owner: 622 (xinetd)</code>	<i>owner process PID and name</i>
<code>flags: ON_RUNQ</code>	<i>current slice operating mode</i>
<code>sliceq[RX]: count 3 avail 8</code>	<i>receive slice queue length and limit</i>
<code>sliceq[TX]: count 0 avail 8</code>	<i>transmit slice queue length and limit</i>
<code>nr_sockets: 1</code>	<i>number of sockets influenced by this slice</i>
<code>nr_connections: 1</code>	<i>number of connections influenced by this slice</i>

Below is an example listing from a per-socket file:

<code>/proc/net/ginsu/slice/7/sock/c0ab88de:</code>	
<code>owner: 7</code>	<i>owner slice identifier</i>
<code>fid: 9</code>	<i>DPF filter identifier</i>
<code>mark: 0x0000</code>	<i>ingress traffic queueing mark</i>
<code>filter: [</code>	<i>DPF filter</i>
<code> m[14:8] & 0xf0 == 0x40 &&</code>	
<code> m[20:16] & 0xff3f == 0x0 &&</code>	
<code> m[23:8] & 0xff == 0x6 &&</code>	
<code> m[30:32] & 0xffffffff == 0x100007f &&</code>	
<code> m[36:16] & 0xffff == 0x180</code>	
<code>]</code>	
<code>nr_sk: 1</code>	<i>number of associated low-level Linux sockets</i>
<code>socket: sk=cf9e9460 [</code>	<i>low-level Linux socket state</i>
<code> family: 2</code>	
<code> type: 1</code>	
<code> protocol: 6</code>	
<code> refcnt: 1</code>	
<code> rcvbuf: 87380</code>	
<code> sndbuf: 16384</code>	
<code> rmem_alloc: 0</code>	
<code> wmem_alloc: 0</code>	
<code> wmem_queued: 0</code>	
<code> receive_queue_len: 0</code>	
<code> write_queue_len: 0</code>	
<code>]</code>	

3. Demonstration

Our demonstration illustrates a typical application of a WebShield-like device as a boundary security gateway with a network-accessible management interface. We show how such a device can be used to protect a subnet by inspecting web and messaging traffic, but must often be “over engineered” (provisioned far in excess of typical capacity) to guarantee service levels. For instance, the demonstration shows that a gateway device can not easily ensure service levels to satisfy both client throughput and management interaction while under heavy loads.

The benefit of the GINSU processing, as demonstrated, reveals that for a given level of hostile (or unwarranted) traffic, the same hardware can appear both more responsive and more tolerant of load spikes. Additionally we demonstrate how, without the GINSU slice isolation features, hostile traffic can adversely affect traffic through the gateway and interfere with a protected client’s use of the network. GINSU, through its use of “Lazy Receiver Processing” and “Per-Slice Queues” is able to efficiently and effectively shed excess traffic based on administrator-applied limits and reservations, before that traffic is able to consume sparse resources on the gateway device

4. Future Work

The Guaranteed Internet Stack Utilization (GINSU) project comes to the FTN program via the ATIAS Survivable Wired and Wireless Infrastructure for the Military (SWWIM) Focused Research Topic. GINSU seeks to guarantee network accessibility by an end-host, even in the event of an attempted denial of service attack. To provide this guarantee of accessibility, we augment an existing operating system’s network stack and kernel with fine-grained resource monitoring. We provide mechanisms for reserving and/or limiting scarce resources based on the ultimate consumer of those resources. We enable attribution of resources based on a rich collection of packet, user, and process attributes. The combination of source partitioning and attribution gives administrators considerable flexibility and power in determining how his system is to be used.

GINSU technology has been proposed in three upcoming research projects. We feel that GINSU brings a powerful policy enforcement and resource tracking mechanism to these projects, and the integration of GINSU into larger, enterprise-scale systems adds a number of useful extensions to the GINSU feature set.

5. References

- [1] P.Druschel and G. Banga, “Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems “, in *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*”, Oct. 1996, pp 261-275, USENIX
- [2] Dawson R. Engler, M. Frans Kaashoek. 1996. Dynamic Packet Filters – “DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation”
- [3] Roger Knobbe, Andrew Purtell, Sept. 2003. “GINSU Administrative Cookbook”.
- [4] Roger Knobbe, Andrew Purtell, Sept 2003. “GINSU As Built Specification”.